PCIe Verification in a SystemC environment using the Cadence VIP

CDN LIVE℠

Cadence User Conference 2014

EMEA – Munich, Germany—May 19-21

25 / 04 / 2014

Maarten de Vries

Verification team leader

© Bull

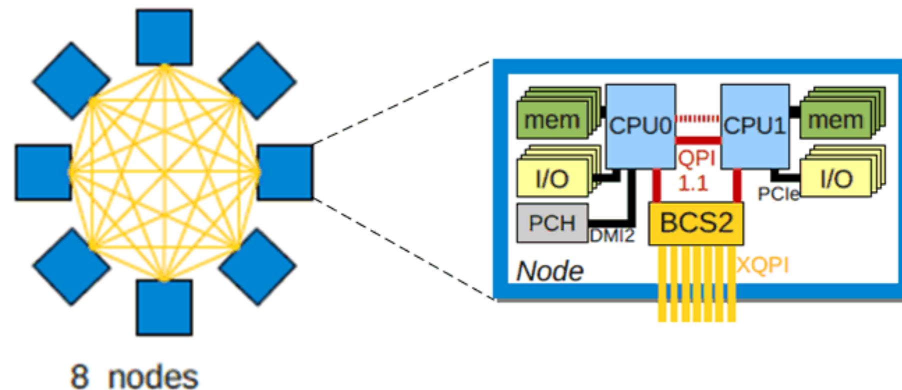
SUMMARY

- BULL : Asic development for HPC
- BULL's next chip will use PCIexpress
- Our PCIe-gen3 verification requirements
- Bull's Verification flow
- PureSpec PCIe VIP integration into GSE
- Difficulties we faced
- Software experience: using the Denali API
- Verification challenges
- Next steps

© Bull, 2013

BULL : Asic development for HPC

- BULL develops « in-house » a few ASIC for High Performance Computing and Data Centers.
- What is HPC ?
  - Computer systems providing peta-flop performances (10^{15} FLOPs)
  - High reliability
  - Large memory foot-print
  - Applications: scientific simulations , weather forecast, big database, database in memory, etc…
- Bull develops ASIC for the high-end computer market.
- BCS2: an example ASIC developed by BULL
  - Allows to interconnect 8 CPU « modules » together
  - 2 socket / module
  - 15 cores / socket
  - => 8 x 2 X 15 cores = 240 cores linked together through a cache-coherent interconnect.
  - 3 TByte DRAM/socket => 48 TByte DRAM managed by an octo-module.



© Bull, 2013

BULL's next chip… uses PCIexpress

- PCIexpress is one of the main interfaces on this ASIC
  - Must be Gen-3 capable,
  - 16 x , which means 16 lanes:
    - 1 Lane = a full-duplex connection (1 Tx, 1 Rx) = a differential serial pair in each direction.
  - 8 Gbit/s/lane => 16 Gbyte/s throughput (in each direction) .

- We have chosen a 3rd party (IP provider) PCIe interface
  - So called « End-Point »,
  - Must work with all possible PCIe switches and « Root-complex ».
  - PCIe allows for many configuration possibilities:
    - A PCIe compliant device doesn't need to support the full standard!
    - Still, many configuration parameters have to be carefully selected.
  - The lane interface requires a high-speed PHY, which is also a 3rd party IP!
    - PHY is at the analog/digital interface and is sensitive to chosen silicon process.
- For all those reasons, verification of such an interface is a challenge!

© Bull, 2013

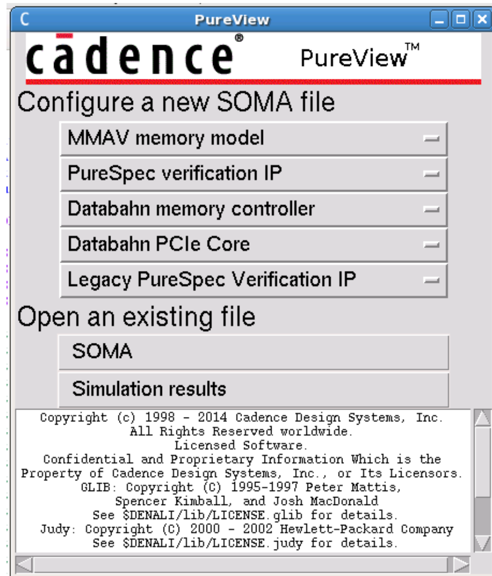
Our PCIe-gen3 verification requirements

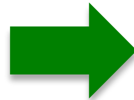
- We need a reliable PCIe Root Complex cycle-accurate VIP
  - Will be used to verify the PCIe End-Point IP (ASIC interface side)
  - Must be flexible:
    - Many parameters : we support only subset of all features some of which are not all clear at the beginning of the project,
  - Must be scalable
    - Number of lanes can change.
  - BULL uses a verification environment using SystemC. The PCIe VIP models should be available directly in SystemC
  - We require different types of test-benches:
    - One bypassing the PHY interface : connection direcly from MAC to MAC
    - One using the full PHY interface: more accurate but slower simulation speed.
    - One reduced version for FPGA simulation (less lanes than 16)
- PureSpec PCIe VIP from Cadence has all requirements!
  - Flexibility, scalability, different HDL languages support (incl. SystemC).
  - PureSpec PCIe VIP has in fact many more good qualities as we will see…

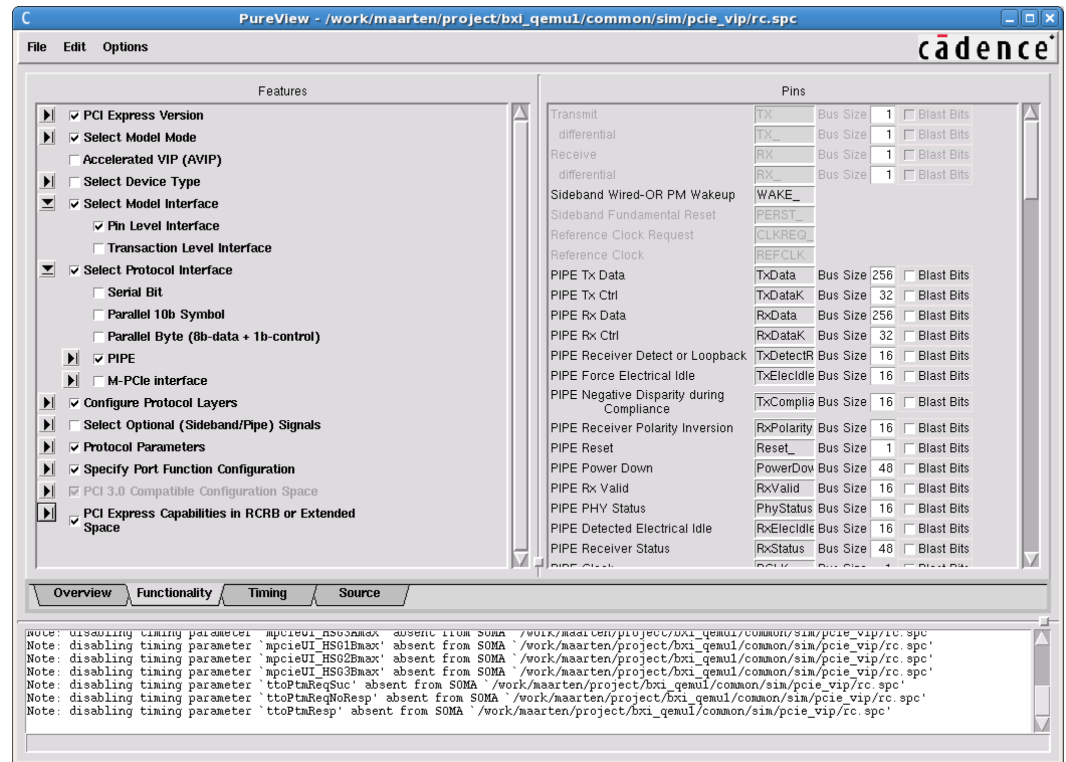
© Bull, 2013

Our PCIe-gen3 verification requirements (cont'd)

- PureView
  - Allows to easily modify the PCIe model (RC, EP-Monitor) by providing all parameters settings through a GUI dashboard:
- Bull decided to choose Cadence PurSpec PCIe VIP.





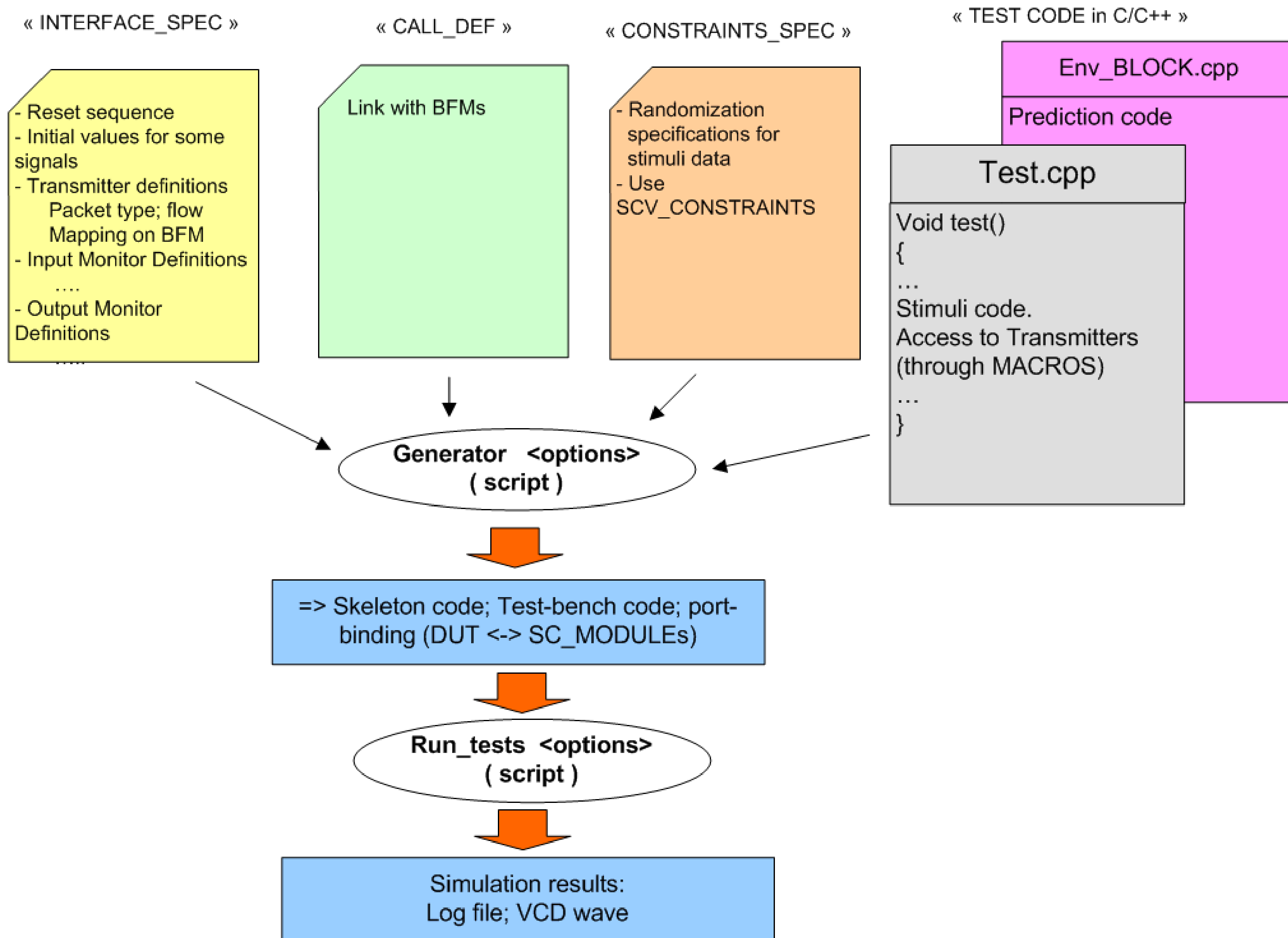


BULL's Verification flow

- To integrate the PureSpec PCIe VIP in our flow, some explanations are needed on BULL's verification flow.
- Flow called « Generic Simulation Environment » (GSE).
  - Tool developed by BULL and used with success on various ASIC projects.
  - Tool based upon scripts and Makefile and allowing to build a SystemC/HDL co-simulation test-bench.
    - DUT in HDL will be SystemC wrapped!
    - Test bench will instantiate Transmitters and Monitors as SC_MODULE based on a set of input-specification files
    - Port binding to Transmitters and Monitors is done automatically based on the same.
    - Manage clock generation and reset.
  - Generation of skeleton C++ code from which the verification engineer can:
    - Drive stimuli through an object oriented programming interface.
      - Have access to constrained randomization of inputs by using the SCV library.
    - Implement all the code to model the DUT behaviour and create the predictions and / or scoreboard.

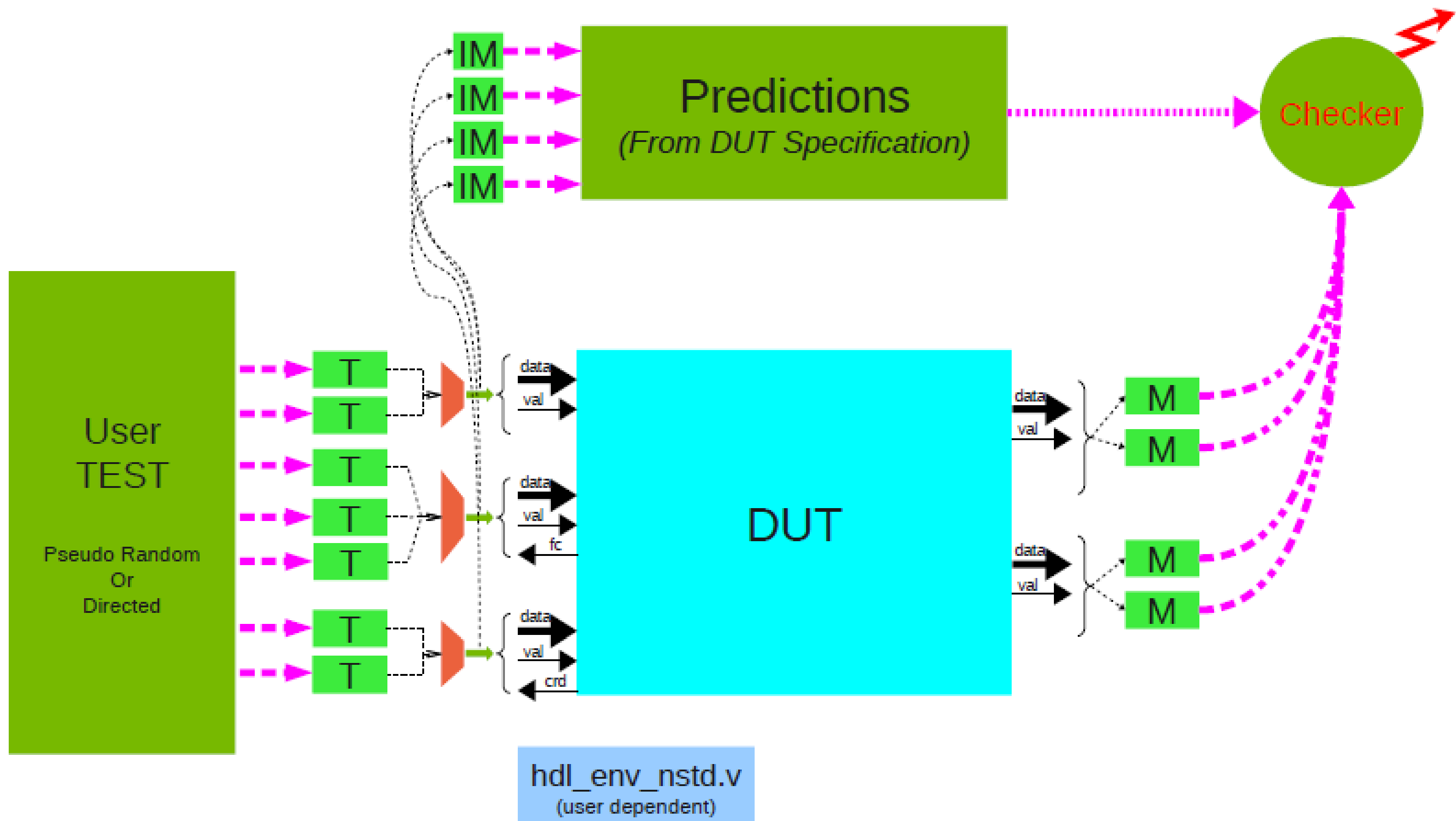
© Bull, 2013

BULL's Verification flow: GSE in a nutshell



© Bull, 2013

BULL's Verification flow: GSE co-simulation test-bench



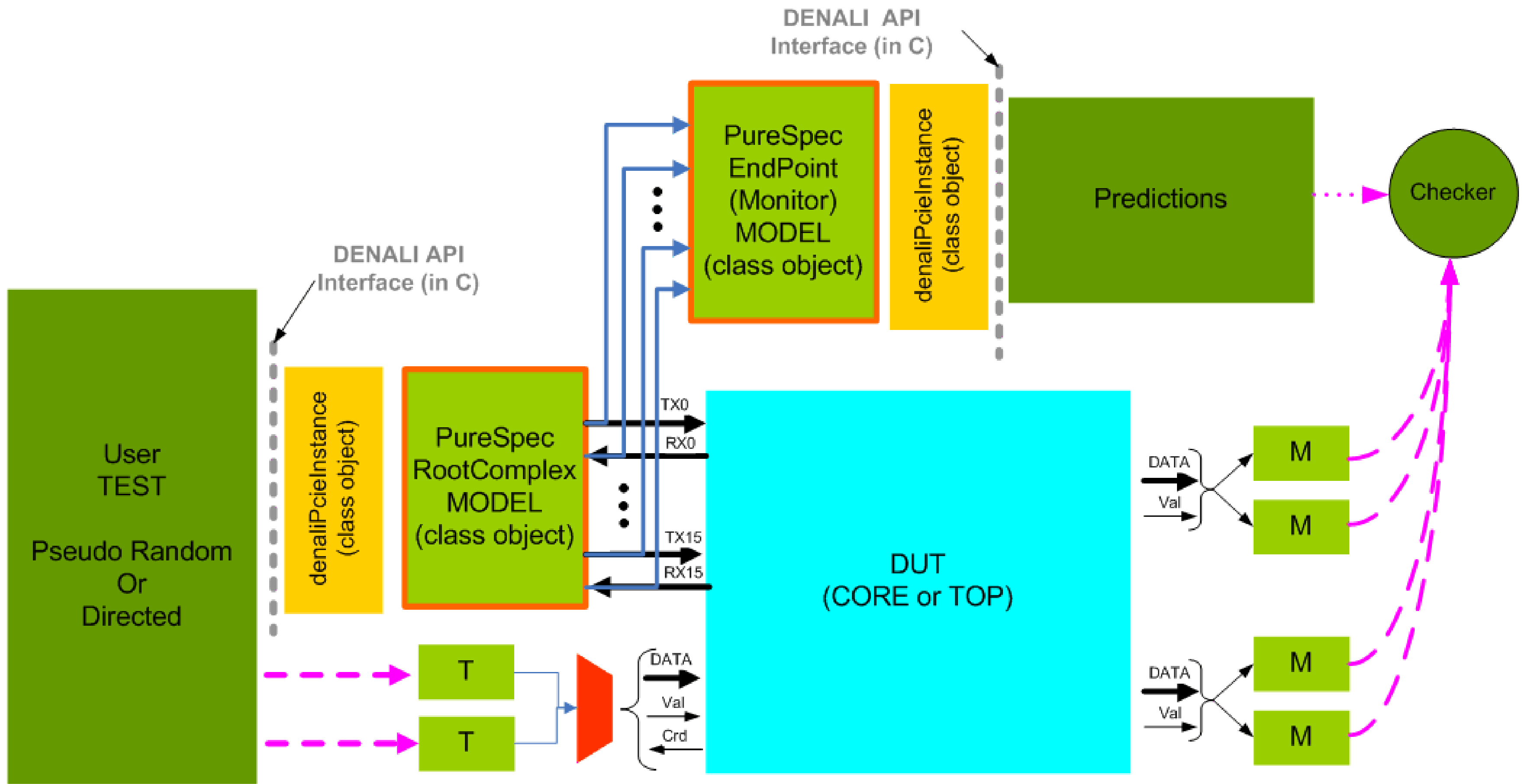
N.B: The green boxes are SystemC SC_MODULE or C++ class objects,
The blue box, is the DUT HDL wrapped in a SystemC SC_MODULE.

© Bull, 2013

PureSpec PCIe VIP Integration into GSE

- What we had to do to integrate the VIP into a GSE test-bench
  - Modify some of the scripts to integrate a custom SC_MODULE
    - GSE flow must remain unchanged !
    - => the instantiation will be controlled through a new option : `-vip=pcie`
    - Denali Model will be instantiated statically in constructor code.
  - Solve interface adaptation between Root-Complex signals and End-Point (IP = DUT side) signals (for the PIPE interface bench):
    - Most signals have a direct match.
    - Some signals were different ! Or different in size (expl: TxDetectRx : shared or per lane: selection through SOMA file with PureView)
      => We had to develop a simple « glue » logic between both worlds.
  - Learn how to access the VIP models from our « `void test() {…}` » code:
    - Denali VIP model can not be accessed directly.
    - Needs instantiation of a « proxy » object of class « denaliPcieInstance »:
      `rc_ = new denaliPcieInstance (« model_name », rcCbFunc);`
    - Proxy object is instantiated dynamically (during run time).

PureSpec PCIe VIP Integration into GSE (cont'd)



© Bull, 2013

Difficulties we faced

- Using the PIPE interface
  - PIPE = **P**HY **I**nterface for the **P**CI **E**xpress Architecture.
    => bypass the PHY on both EP and RC, and connect the MAC layers (parallel data bus and signaling).
    => This also means many signals to manage and understand…
  - The SystemC model generated with PureView when selecting the PIPE i/f mode, happened to be bugged, but Cadence provided a patch.
  - Some problem with scrambling mode => Cadence had to provide another patch.
  - Some signals driven with 'X' => Cadence had to provide a patch for the PIPE interface.
  - Discovering traffic generated automatically related to setup MSI-X tables!
  - More than 10 Cadence Case tickets opened by our Verifcation engineers.
  - Our PCIe End-Point IP didn't work immediately as expected (TLP traffic not accepted by EP). We also had to fine-tune the EP IP.
  - The VIP allows to generate very detailed log files:
    - All the exchanges occurring on the PCIe link.
    - Requires some habit to learn to understand the meaning of some messages, warning, or errors.
  - Addressing is quite tricky!
    - Crossing 4 K boundaries will cause so-called malformed TLP…
    - BAR programming: contains both address and size (number of LSB to 0) !
  - PCIe specification is huge (850 pages), and some difficulties come from our lack of detailed knowledge! Learning new topics every day!

© Bull, 2013

Software experience: using the Denali API

- The initialization software sequence required before transactions may be started:

BASIC SOFTWARE INIT SEQUENCE

```
If (-1 == denaliPcieInit()) {
     cout << "Failed to Init ddvapi" << endl;
}

rc_ = new denaliPcieInstance("my_rc", rcCbFunc);

rc_->addcb(PCIE_CB_PL_TX_start_packet);
rc_->addcb(PCIE_CB_TL_RX_packet);
rc_->addcb(PCIE_CB_TL_TX_packet);
... Etc …

denaliPcieGetIdByName("my_rc(cfg_0_0)", &id_rc_cfg_0);
denaliPcieGetIdByName("my_rc(mem_0_0_0)", &id_rc_mem_0_0_0);
denaliPcieGetIdByName("my_rc(p_0.cfg_0_0)", &id_ep_cfg_0);

while (((*device_state) & 0x0F) != 0x2) {
  denaliPcieRead(id_rc_cfg_0, PCIE_REG_DEN_DEV_ST, &device_state);
}

Write_Reg(id_ep_cfg_0,PCIE_REG0_BASE_0,some_address);
Write_Reg(id_ep_cfg_0, PCIE_REG_DEN_MEM_CFG,(PCIE_MEM_OP_CFGWR |
(PCIE_REG0_BASE_0 << 4)));
...

// Ready to send TLP packets
   rc_->transAdd(pkt,0,DENALI_ARG_trans_append);
```

Model and API Initialization

Dynamic allocation of « proxy » object to allow access to the VIP model + give Call-Back func name

Register Call-Back « reasons »: rcCbFunc() will be called when the reaon happens.

Obtain an « id » for: the RC, the RC memory, the EP (required to access these memory areas)

Wait for the PCIe link to be ready (L0 state of LTSSM state machine)

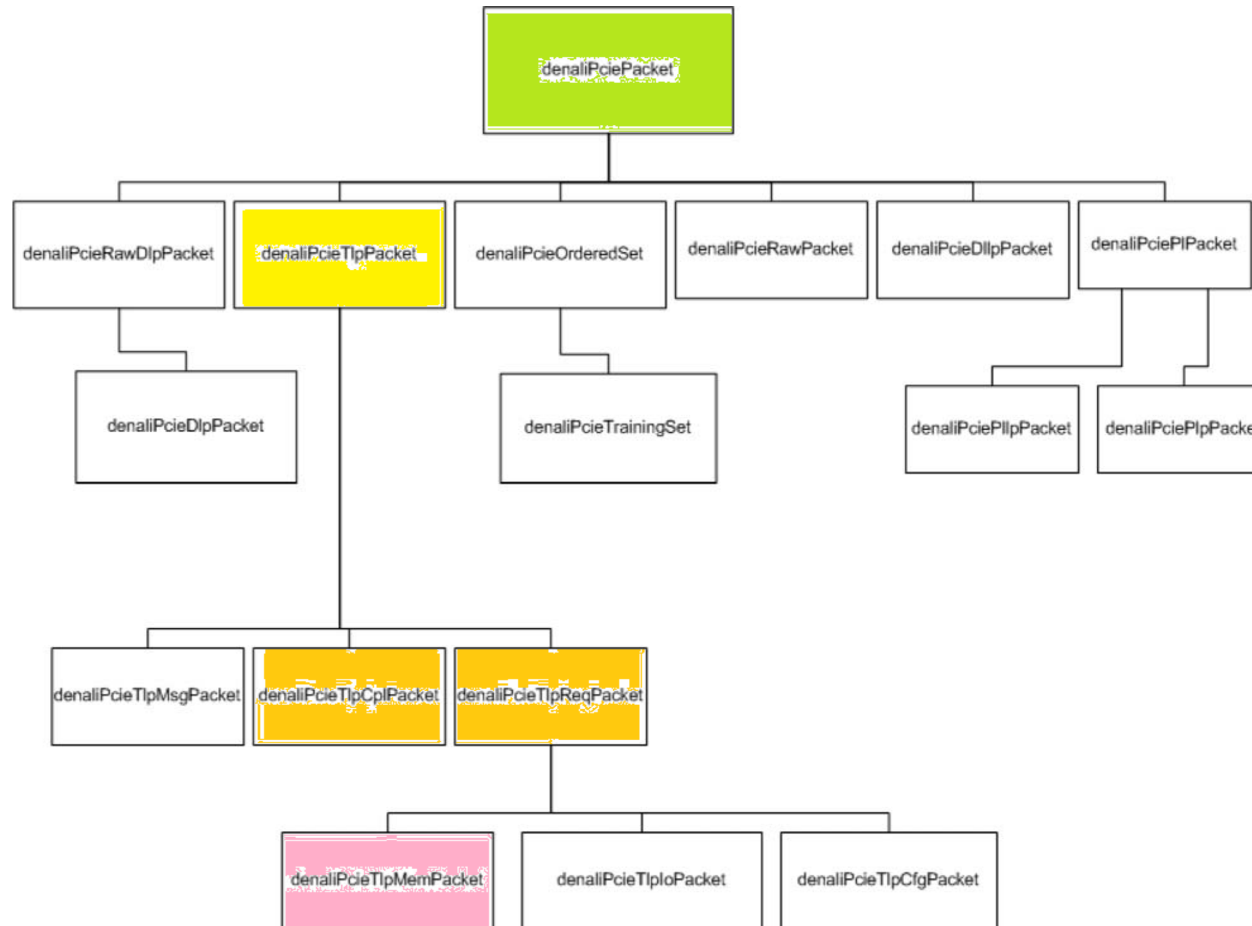
Re-program the BAR registers in the EP (if necessary) (change address and size)

The VIP is ready to be used: TLP transactions may be send.

© Bull, 2013

Software experience: using the Denali API (cont'd)

Denali PCIe Packet Class diagram: required to know when you send for instance TLP or receive them, and for Completions. We mainly used the coloured ones:



© Bull, 2013

Software experience: using the Denali API (cont'd)

Summary of interfaces (Verification SW / Denali RC VIP)

- VIP → DUT(CORE)

  ```
  rc_->transAdd(tlpRd, 0, );
  ```
  Where « tlpRd » is a pointer on a TLP packet type: `denaliPcieTlpMemPacket`
  See figure on slide 14 (Denali PCIe Packet Class diagram).
- Completions to Read requests will be handled by the Callback function (slide 16)

- Instantiation will pass type (Rd | Wr) through constructor arg:
  for Read:
  ```
  denaliPcieTlpMemPacket   tlpRd(DENALI_PCIE_TL_MRd_32);
  ```
  for Write:
  ```
  denaliPcieTlpMemPacket   tlpWr(DENALI_PCIE_TL_MWr_32);
  ```

  Then packet fields must be given one by one based on target address, length, …etc.
  ```
  tlpRd.setPktDelay(0);
  tlpRd.setRequesterId (ReqId);
  tlpRd.setTransactionIdTag(some_TransactionIdTag);
  tlpRd.setLength (some_length);
  tlpRd.setLastBe(LastBe);
  tlpRd.setFirstBe(FirstBe);
  tlpRd.setAddress(Addr);
  tlpRd.setAddressHigh(AddrHigh);
  tlpWr.setPayload(&payload_lxbar);    // For Write: payload ptr
                                       // prepared elsewhere
  ```

© Bull, 2013

Software experience: using the Denali API (cont'd)

- Summary of interfaces (Verification SW / Denali RC VIP)
  - DUT(CORE) → VIP
  - 1) For Writes initiated by the DUT to VIP:

A callback function mechanism is provided by the VIP.
When a TLP Write Request is received from the DUT, the **callback function** will be called. We have to write the code for it like this:

```
int rcCbFunc (int id, DENALIhandleT transHandle, int
reasonID)
{
switch (reasonID)
     {
        case PCIE_CB_TL_user_queue_enter:   // VIP → DUT
   ...              // Used to check that Completions have been send back to ASIC.
        case PCIE_CB_TL_TX_packet :   // VIP → DUT
   ...                 // Used to spy on Configuration packets or inject errors.
        case PCIE_CB_TL_RX_packet :
   ...                   // Code here to handle DUT→VIP write requests
                         // AND: Completions from previous VIP→DUT Read Req.
        default :
}
```

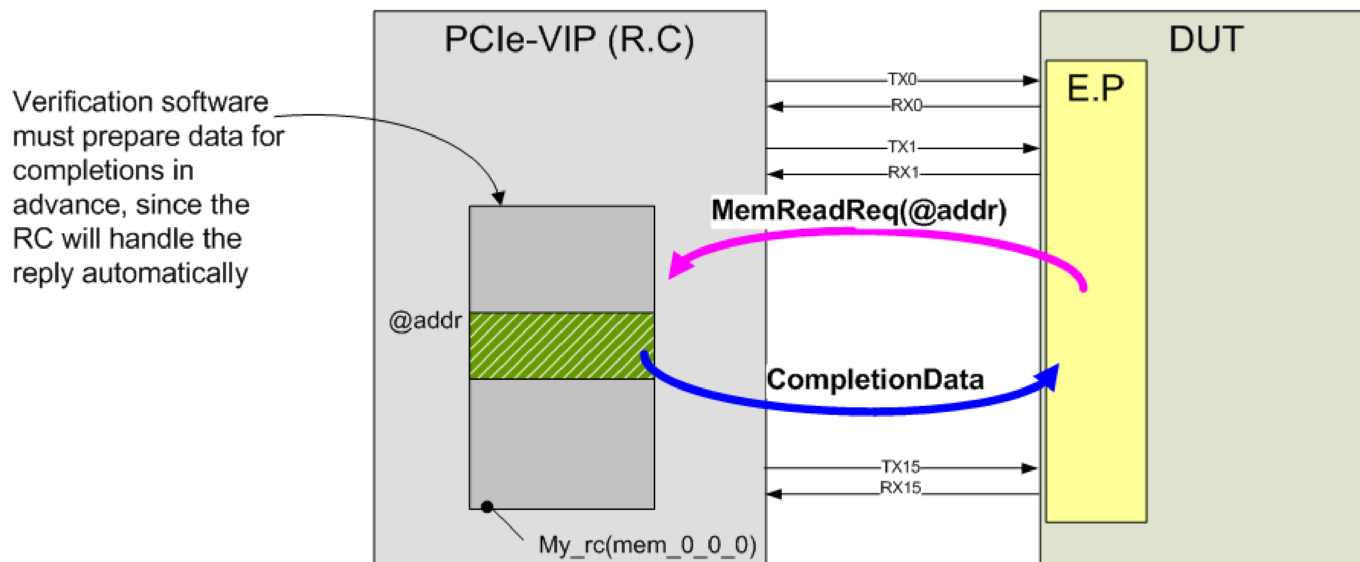
© Bull, 2013

Software experience: using the Denali API (cont'd)

Summary of interfaces (Verification SW / Denali RC VIP)

- DUT(CORE) → VIP
- 2) For Reads from DUT → VIP

The VIP has its own memory area instantiation.
When a TLP Read-req is received by the VIP (from the DUT), the VIP will return automatically 1 or more completions with the data read at the given address in this VIP memory area.



© Bull, 2013

Verification Challenges

- CORE / TOP Verification code must not rewrite application software!
- PCIe TL/Denali-API only used to write to / read from the DUT, and to be notified when the DUT has information for the VIP (Completions to MemRead, Writes / Reads initiated by the DUT)
- We very much rely on PCIe link to work once the so-called LTSSM/L0 state reached.
- We use a mix of directed test-cases and constrained random.
- However, remains close to how application SW would behave.
- We cannot directly reuse application SW, since RTL simulation can only focus on very short sequences => ~30 mn to several hours for 1 test case.
- Complex DUT architecture requires 4 full time Verification Engineers to develop and test the C/C++ code.
- 1 of the 4 is responsible for the integration.

© Bull, 2013

Next steps

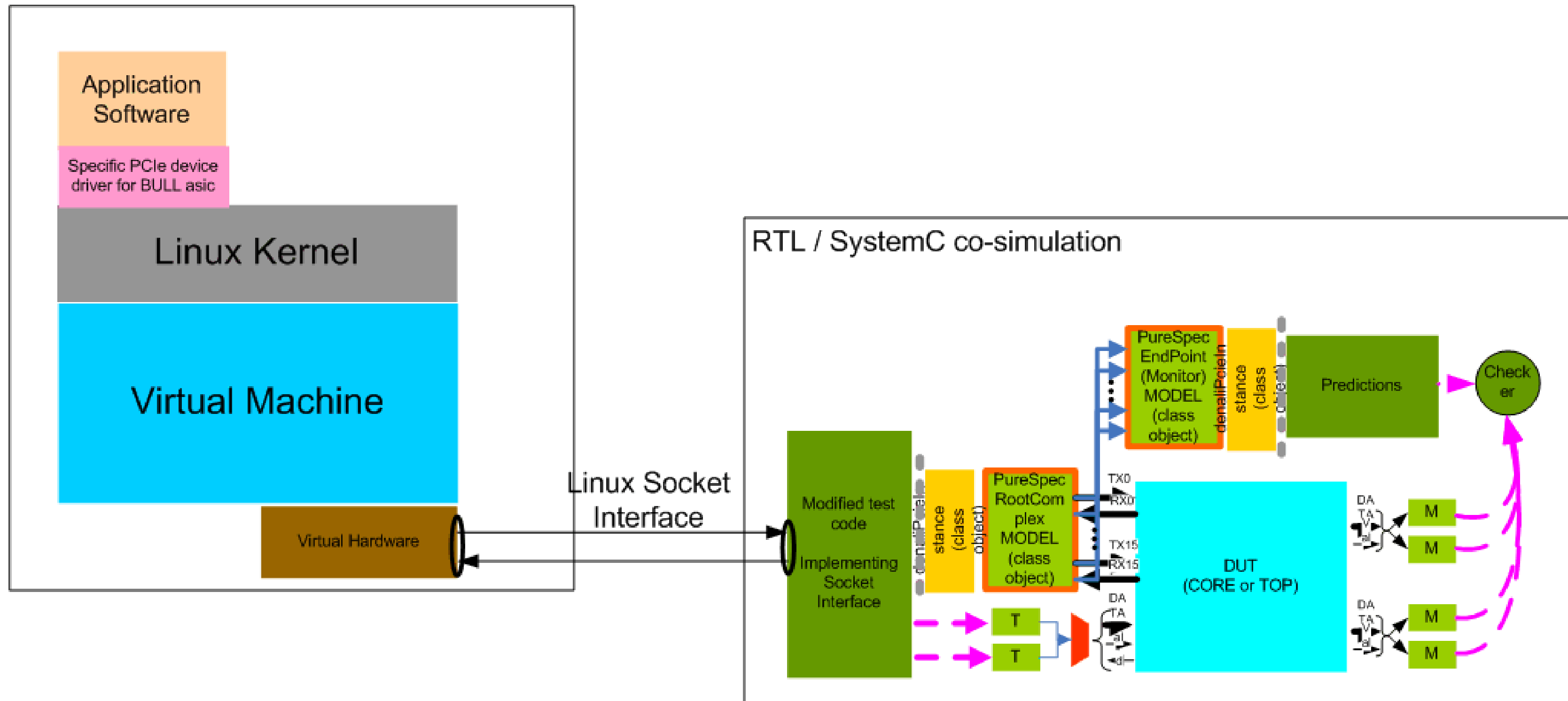
HW / SW co-verification

- We are experimenting with co-verification, since this chip has a strong intercation with application software.
- Final chip will be controlled through a Linux-based PCIe device driver.
- Complex application software will interact with chip, making verification an even greater challenge. One solution is to use a HW/SW co-verification bench, based on a virtual machine.
- Make use of linux-socket interface communications to plug on to the RTL/SystemC CORE simulation.
- Benefit of SystemC environment is obvious.
  - Straightforward interaction with Linux system api.

© Bull, 2013

Next steps (cont'd)

HW/SW co-verification test bench overview:



Architect of an Open World™